SUPERCHARGE YOUR DATA MODELING

The DynamoDB Book

Operations Supplement

Alex DeBrie

Operations with DynamoDB

Alex DeBrie

Version 1.0, 2020-04-03

Table of Contents

Chapter 1. Operations on your DynamoDB Tables

In the very first chapter of The DynamoDB Book, I mention that DynamoDB is a *fully-managed database*. AWS provides it to you as a service, and they handle all instance provisioning, software updates, hardware failovers, and more.

But that doesn't mean you're completely off the hook for operations. There are still some important areas for which you're responsible. In this additional booklet, we'll cover operations in DynamoDB.

We'll start with authorization. We'll see how DynamoDB uses the powerful AWS Identity and Access Management (IAM) system for authentication and authorization. With AWS IAM, you get role-based authentication, fine-grained authorization, and an easier overall authorization story than traditional databases.

Second, we'll look at how to provision your DynamoDB tables. While most people provision their first table in the AWS console with a friendly point-and-click wizard, this isn't the recommended option for production use cases. We'll see why you want to use infrastructure-as-code for provisioning and managing your tables and IAM permissions, and we'll walk through some examples by using AWS CloudFormation.

Third, we'll talk about DynamoDB pricing. The pricing model with DynamoDB is unique in the database world. Rather than paying for some resources (CPU, RAM, disk) and trying to guess how that translates to query performance, you pay for your read and write operations directly. We'll cover the two billing modes of DynamoDB (provisioned vs. on-demand) and which one you should choose. We'll also see that the best way to save money is usually not related to your billing mode choice.

Finally, we'll cover disaster management. Data is the most important part of your application, and you need a plan when everything goes sideways. DynamoDB has two options for backing up your data, including one that allows you to restore your DynamoDB table to any specific second in the past 35 days! We'll see how to think about your backup options and how to restore your data.

DynamoDB isn't "NoOps", but it greatly reduces the operations burden. You can see how the 'born-in-the-cloud' nature of DynamoDB has allowed the DynamoDB service team to rethink how operations work. Whether it's fine-grained authorization, infrastructure-as-code provisioning, on-demand pricing, or point-in-time restores, DynamoDB is pushing the envelope on making a database that's easier to use and operate.

Chapter 2. DynamoDB Authorization

We're going to start our discussion of DynamoDB operations on the topic of authorization because it's the most important operational concern with respect to data. Your users are trusting that you'll safeguard their data. In many cases, it'd be better for the data to be lost than to be accessed by the wrong person.

Fortunately, DynamoDB has a robust authorization model for keeping your data safe. You can limit calls to specific tables, specific actions, or even specific items or attributes. Further, the integration with AWS IAM greatly simplifies credential management when using AWS compute.

In this chapter, we'll discuss the basics of authorization and the AWS IAM system. Then we'll talk about IAM policies and least privilege access. Finally, we'll touch on some advanced topics like the different DynamoDB resource types and using conditions in IAM policies.

2.1. Basics of Authorization

All authorization for DynamoDB requests goes through the AWS Identity and Access Management (AWS IAM) system. By using IAM, you get access to a rich authorization schema that allows for fine-grained permissions, cross-

account permissions, temporary credentials, and more. We will review the basics of AWS IAM before seeing how it applies to DynamoDB.

2.1.1. Authentication and Authorization

There is a two-step process whenever you make a request to AWS. First, your request is authenticated to determine the entity that is making the request. Second, the request is checked to ensure the caller has the proper authorization to perform the requested action.

For nearly all requests to AWS, you must be authenticated as some sort of IAM entity. You can authenticate using a username and password if you are signing in to the AWS console. More commonly, you will interact with the AWS API via the command line or an AWS SDK, in which case you will authenticate with an AWS Access Key and an AWS Secret Key.

When you are authenticated, AWS will determine the *entity* making the request. An AWS identity can be an IAM user, a federated user from an existing system in your organization, or an assumed IAM role. IAM users and roles are discussed in further detail below.

Once the request is authenticated, then AWS IAM will ensure the authenticated IAM entity has the authorization to perform the requested action. Each action in AWS is a combination of an AWS service and the service action. For example, the action to allow the GetItem operation on a

DynamoDB table is `dynamodb:GetItem`. The action to allow a PutObject request in an S3 bucket is `s3:PutObject`.

AWS IAM will confirm that the authenticated entity has permission to perform the requested action on the resource requested. This involves checking any policies associated with that entity to see if the action is allowed as well as ensuring there are no policies that specifically disallow the requested action. An IAM policy is a JSON document attached to a particular identity that defines the permissions granted to that identity.

2.1.2. IAM Identities and Entities

To finalize the basics of AWS IAM, let's cover IAM identities and entities. They contain some concepts that appear to overlap, and the difference is subtle.

When you are handling access management in AWS IAM, you will do so by assigning IAM policies to different identities. These policies state which permissions are granted or denied to a particular identity.

There are three categories of identities to which you can assign permissions:

- **Users:** This represents a user within your AWS account. If it's a human user, it may include a username and password for signing in to the AWS console. It likely has some AWS credentials for authenticating via AWS API calls. These credentials are long-lived and only expire when explicitly told to expire.

- **Groups:** A group is a container for IAM users that have similar permissions. You can attach IAM policies to a group and then add users to the group to allow for access management across a large number of users. Groups themselves do not have credentials; only users that belong to groups have credentials.
- **Roles:** A role is another type of entity in your AWS account. However, rather than being associated with a specific user, a role can be *assumed* by others. A role may be assumed by a user, or it may be assumed by an AWS service, such as an EC2 instance or a Lambda function. Roles do not have permanent credentials. Rather, they use temporary credentials that expire after a given time.

While *identities* are the objects that have IAM permissions attached, *entities* are the objects that are authenticated by AWS IAM. There are three types of entities as well:

- **IAM User:** You can use permanent credentials from an IAM user to authenticate and use the permissions for that user.
- **Federated User:** If you have an existing authentication system, you can federate those identities into AWS rather than creating new IAM users for everyone.
- **Assumed IAM Roles:** If an identity has assumed a role, credentials for the assumed role are used for AWS requests, and that role is authenticated by AWS. During the authorization phase, it looks to the permissions granted to the IAM role in determining whether the request is allowed.

2.1.3. IAM roles with your application

While you may use IAM users with long-lived credentials to develop locally or sign in to the AWS console to view your DynamoDB tables, most of your DynamoDB access should be through IAM role attached to AWS compute. If you are using Amazon EC2 or AWS Lambda for your compute, AWS makes it simple to use roles. You simply attach the role to your EC2 instance or Lambda function, and that compute will have temporary credentials in the AWS environment.

These temporary credentials work seamlessly with all AWS SDKs. For example, if you were using permanent IAM credentials from an IAM User, your SDK client initialization would need to look as follows:

```
import boto3

client = boto3.client(
    'dynamodb',
    aws_access_key_id='<yourAccessKey>',
    aws_secret_access_key='<yourSecretKey>'
)

client.get_item(...)
```

Notice that this method requires you to manage your AWS credentials. You need to explicitly pass them into your client initialization. You would also be in charge of credential rotation to limit the risk if your credentials were accidentally leaked.

If you are using IAM roles with your AWS compute, then client initialization looks as follows:

```
import boto3

client = boto3.client('dynamodb')

client.get_item(...)
```

There's no explicit passing of credentials. The `boto3` SDK (or relevant SDK for your programming language) knows how to retrieve credentials from the attached IAM role. These credentials are temporary and will be rotated automatically for you. It's easier and safer than using permanent credentials yourself.

2.2. IAM policies & least privilege access

We discussed above that you grant permissions to IAM identities using IAM policies. In this section, we'll look at some example policies to see how to grant permissions.

One of the most important principles around IAM policices is that you should adhere to *least privilege access*. As you are writing policies, ask yourself "What is the least amount of access this identity needs to accomplish the goal?" Let's see this in action.

Below is an example of an IAM permission statement that is contained in an IAM policy.

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Action": [
        - "dynamodb:GetItem",
        - "dynamodb:PutItem"
    ],
    "Resource": [
        - "arn:aws:dynamodb:us-east-2:123456789012:table/SaaSApp"
    ]
  }
}
```

There are three elements to this IAM permission statement:

- **Effect:** The effect states how this statement should be applied. "Allow" means to allow the IAM identity to perform the listed actions on the listed resources in the statement. "Deny" means to specifically *disallow* the actions on the resources. By default, all actions are denied unless given specific "Allow" permission. Further, an explicit "Deny" permission will override an explicit "Allow" permission.
- **Action:** This describes the actions allowed in the statement. You can provide a list of allowed actions. Each action will be in the format of `<service>:<operation>`, such as `dynamodb:GetItem` or `s3:PutObject`. You may use wildcards, such as `dynamodb:*`, to allow all actions in DynamoDB, but you should avoid where possible. It is unlikely you need *every* action in DynamoDB. Aim for least privilege access!
- **Resource:** The resource property states to which resources the statement should apply. This allows you to scope your permissions to a single DynamoDB table or

S3 bucket rather than all tables or buckets in your account. The values in this array are different ARNs (Amazon Resource Names) that identify unique resources in your account. In this example, we are allowing the two DynamoDB actions to be performed on a specific table (`SaaSApp`, which is located in the `us-east-2` region) in our account.

Be intentional when writing these IAM policies. You may be tempted to use wildcards to save time, but this can result in leaked access or undesirable effects on your resources. Limit the access you grant by listing the specific actions you want on the specific resources in your application.

2.3. Advanced concepts

Before we wrap up this chapter, I want to cover two advanced concepts in IAM that you may need when working with DynamoDB. Those concepts are:

- The resource types in DynamoDB
- IAM conditions for DynamoDB

2.3.1. DynamoDB resource types

In our example IAM statement above, we showed how to apply a statement to a specific DynamoDB table. Let's take a look at the example ARN we used and how to break down the various elements:

```
"arn:aws:dynamodb:us-east-2:123456789012:table/SaaSApp"
"arn:aws:<service>:<region>:<account>:<resourceType>/<resourceName>"
```

You can see that each ARN can be constructed from its constituent elements. There is a service name, a region name, an account number, a resource type ("table"), and a resource name ("SaaSApp").

When modeling with DynamoDB in your application, you will commonly use other resource types: indexes and streams. Those have a slightly different ARN format, so let's look at some examples.

First, a DynamoDB index ARN is as follows:

```
"arn:aws:dynamodb:us-east-2:123456789012:table/SaaSApp/index/UsersIndex"
"arn:aws:<service>:<region>:<account>:<resourceType>/<resourceName>/<subresou
rceType>/<resourceName>"
```

The index ARN is similar to a table ARN with an additional `/index/<IndexName>` tagged onto the end. Note that you need to grant permissions like `dynamodb:Query` to *both* your DynamoDB table *and* your DynamoDB index. Granting the Query permission on the table resource does not also grant permissions to sub-resources on your table like indexes.

Second, a DynamoDB stream ARN looks like the following:

```
"arn:aws:dynamodb:us-east-2:123456789012:table/SaaSApp/stream/2015-05-
11T21:21:33.291"
"arn:aws:<service>:<region>:<account>:<resourceType>/<resourceName>/<subresou
rceType>/<resourceLabel>"
```

The stream ARN is similar to an index ARN. It appends `/stream/<StreamLabel>` to the end of the DynamoDB

table resource.

While these two additional resource types are the most commonly used in data modeling with your application, there are two other ARN formats relevant for DynamoDB: DynamoDB backups and DynamoDB Global Tables. Those ARN formats are shown below.

```
# Backup
"arn:aws:dynamodb:us-east-2:123456789012:table/SaaSApp/backup/01489173575360-
b308cd7d"

# Global Table
"arn:aws:dynamodb::123456789012:global-table/SaaSApp"
```

Notice that the ARN for a Global Table does not include a region because it spans multiple regions.

2.3.2. Conditions

When writing IAM policy statements, you may add Conditions to the statement to provide even more fine-grained access to your tables. These Conditions must be true in order for the statement to apply.

Let's look at two different Condition statements with DynamoDB. First, we can apply the `dynamodb:Attributes` Condition to limit the attributes that a particular identity can read:

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Action": [
        - "dynamodb:GetItem",
        - "dynamodb:PutItem"
    ],
    "Resource": [
        - "arn:aws:dynamodb:us-east-2:123456789012:table/SaaSApp"
    ],
    "Condition": {
        "ForAllValues:StringEquals": {
            "dynamodb:Attributes": [
                "UserName",
                "OrgName"
            ]
        },
        "StringEqualsIfExists": {
            "dynamodb:Select": "SPECIFIC_ATTRIBUTES",
            "dynamodb:ReturnValues": [
                "NONE",
                "UPDATED_OLD",
                "UPDATED_NEW"
            ]
        }
    }
  }
}
```

In our Condition block, we're only allowing this identity to read the `UserName` and `OrgName` attributes from our DynamoDB table. This would prevent this particular identity from reading other, sensitive attributes such as `Admins` (showing the list of administrators for an organization) or `Plan` (showing information about the payment plan for an organization).

A second example of using Conditions is with the following statement:

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Action": [
        - "dynamodb:GetItem",
        - "dynamodb:PutItem"
    ],
    "Resource": [
        - "arn:aws:dynamodb:us-east-2:123456789012:table/SaaSApp"
    ],
    "Condition": {
        "ForAllValues:StringEquals": {
            "dynamodb:LeadingKeys": [
                "${cognito-identity.amazonaws.com:sub}"
            ]
        }
    }
  }
}
```

In this example, we're using the `dynamodb:LeadingKeys` condition. For a user that has authenticated via Cognito, they are only allowed to view DynamoDB items that start with the `sub` property of the Cognito context. This limits a Cognito user to only view the items that belong to them.

2.4. Conclusion

In this chapter, we learned how authentication and authorization are used with DynamoDB. DynamoDB uses the powerful AWS IAM system for handling access management to DynamoDB resources.

There are two main points to remember with DynamoDB access management:

- Use IAM roles attached to AWS compute for easy credential management

- Use least privilege access to provide only the permissions needed for your application

By following these rules, you'll avoid the biggest potential security issues in your application.

Chapter 3. Provisioning your DynamoDB Tables

Before you store any data in DynamoDB, you'll need to create a DynamoDB table. In this chapter, we'll talk about best practices for provisioning your DynamoDB tables. We'll discuss the different options you have and why you should prefer using infrastructure-as-code when provisioning your resources. Next, we'll walk through examples of provisioning DynamoDB tables and related IAM resources with CloudFormation. Finally, we'll discuss a few gaps around infrastructure-as-code with DynamoDB.

3.1. Three methods of provisioning

There are a few different methods you can use to provision and configure your DynamoDB resources. At a high level, the main methods are:

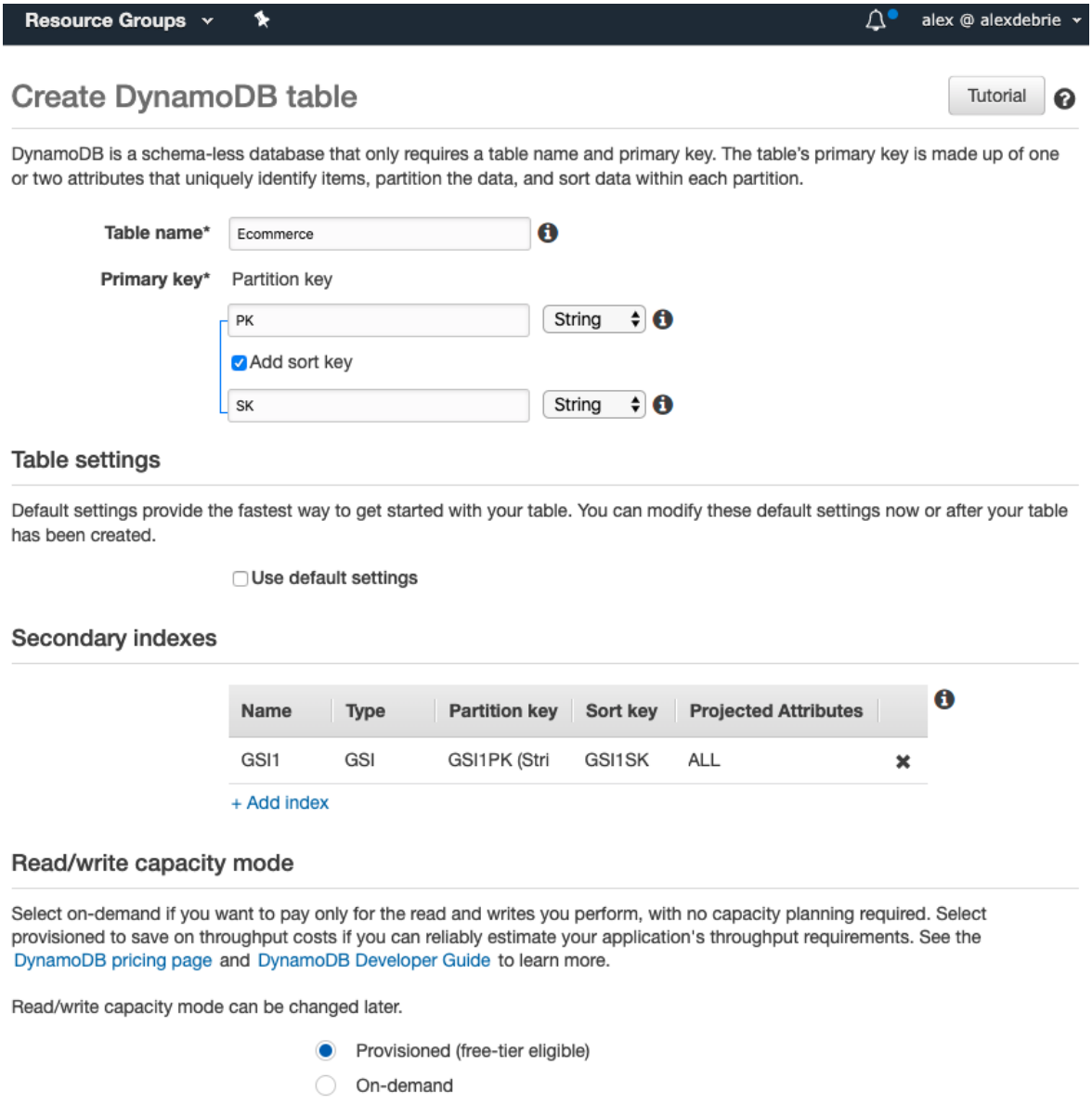
- Using a GUI via the AWS console
- Using an API via the AWS CLI or an SDK
- Using infrastructure-as-code such as CloudFormation or Terraform

Let's walk through each of these and then see why I recommend using infrastructure-as-code.

3.1.1. Using the AWS console

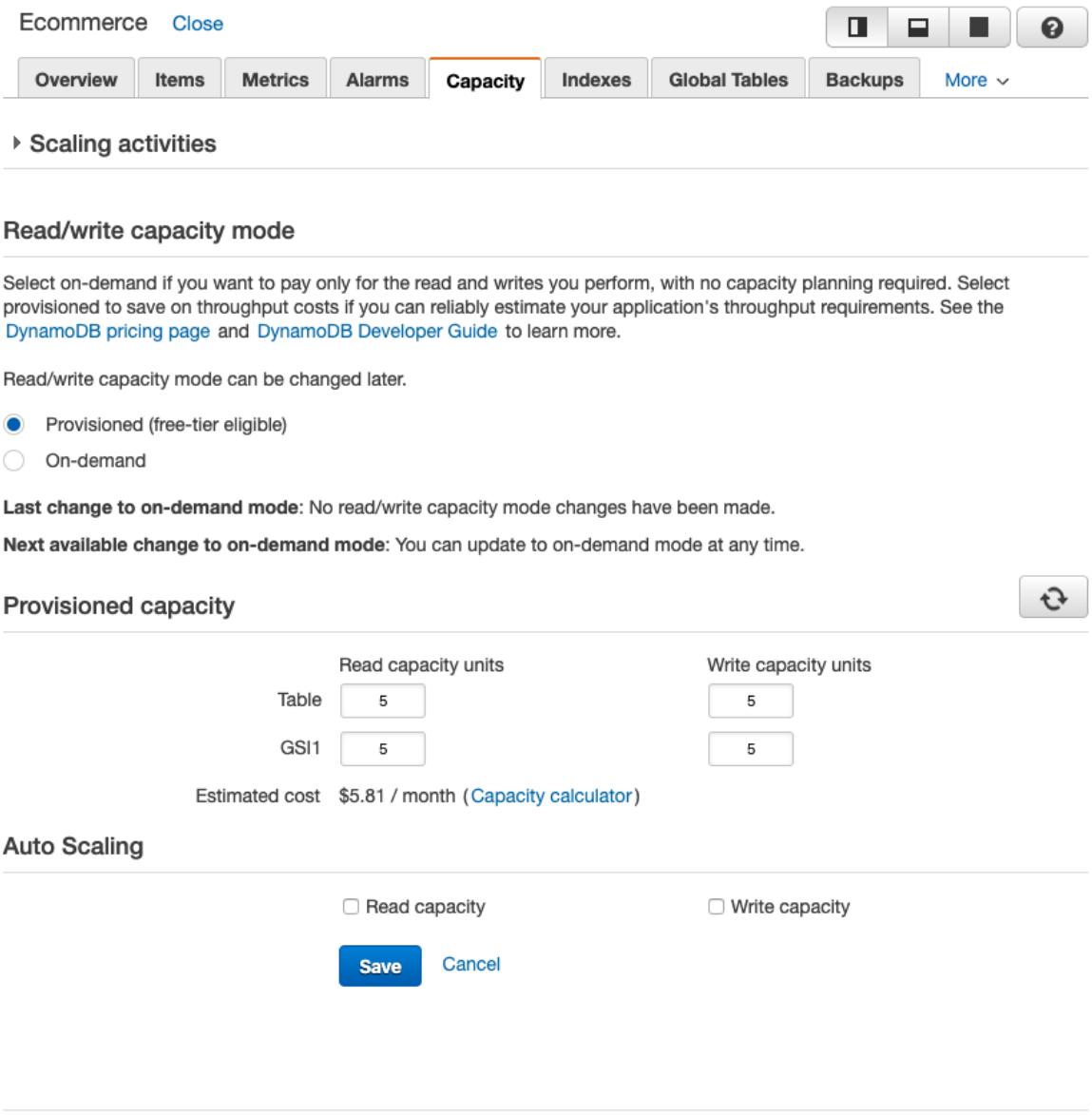
The first option for provisioning your resources is to create or configure your DynamoDB table via a GUI using the AWS console. You will log into the AWS console as an IAM user, then create or update your table through various UI screens.

For example, the wizard to create your table looks as follows:



In this wizard, you add your table name, your primary key structure, and your billing parameters. You can also add secondary indexes or a time-to-live (TTL) configuration as needed.

In addition to the create table flow, you can also update an existing table in the following screen:



This can be used to change your billing settings, and there are similar tabs to add a global secondary index or delete a table.

The nice thing about doing this via the AWS console is that it feels easier. If you're new to DynamoDB, all the settings and configuration options can seem overwhelming. The AWS console wizards help with sensible defaults and link out to additional information where needed.

3.1.2. Using an API via the AWS CLI or an SDK

While the AWS console works for some folks, a lot of AWS users prefer to interact with AWS over an API rather than in a web browser. The most common way to do this is via the AWS CLI, though you can use an AWS SDK with your favorite programming language also.

To provision a DynamoDB table via the AWS CLI, you would enter something like the following into your terminal:

```
aws dynamodb create-table \
    --table-name MovieRoles \
    --attribute-definitions '[
        {
            "AttributeName": "Actor",
            "AttributeType": "S"
        },
        {
            "AttributeName": "Movie",
            "AttributeType": "S"
        }
    ]'
    --key-schema '[
        {
            "AttributeName": "Actor",
            "KeyType": "HASH"
        },
        {
            "AttributeName": "Movie",
            "KeyType": "RANGE"
        }
    ]'
    --provisioned-throughput '{
        "ReadCapacityUnits": 5,
        "WriteCapacityUnits": 5
    }'
```

This creates our MovieRoles table that we've used as an example and provisions 5 read and write capacity units.

Despite the length of the command, that's actually a simple example. You can also add secondary indexes when creating a table via the CLI. This increases the complexity of your command:

```
aws dynamodb create-table \
    --table-name MovieRoles \
    --attribute-definitions '[
        {
            "AttributeName": "Actor",
            "AttributeType": "S"
        },
        {
            "AttributeName": "Movie",
            "AttributeType": "S"
        }
    ]'
    --key-schema '[
        {
            "AttributeName": "Actor",
            "KeyType": "HASH"
        },
        {
            "AttributeName": "Movie",
            "KeyType": "RANGE"
        }
    ]'
    --global-secondary-indexes '[
        {
            "IndexName": "ByMovie",
            "KeySchema": [
                {
                    "AttributeName": "Movie",
                    "KeyType": "HASH"
                },
                {
                    "AttributeName": "Actor",
                    "KeyType": "RANGE"
                }
            ],
            "Projection": {
                "ProjectionType": "ALL"
            },
            "ProvisionedThroughput": {
                "ReadCapacityUnits": 5,
                "WriteCapacityUnits": 5
            }
        }
    ]'
    --provisioned-throughput '{
        "ReadCapacityUnits": 5,
        "WriteCapacityUnits": 5
    }'
```

Others prefer to use an AWS SDK to provision their tables. The comparable CreateTable command in `boto3` would look as follows:

```
resp = client.create_table(
    TableName='MovieRoles',
    AttributeDefinitions=[
        {
            'AttributeName': 'Actor',
            'AttributeType': 'S'
        },
        {
            'AttributeName': 'Movie',
            'AttributeType': 'S'
        }
    ],
    KeySchema=[
        {
            'AttributeName': 'Actor',
            'KeyType': 'HASH'
        },
        {
            'AttributeName': 'Movie',
            'KeyType': 'RANGE'
        }
    ],
    GlobalSecondaryIndexes=[
        {
            "IndexName": "ByMovie",
            "KeySchema": [
                {
                    "AttributeName": "Movie",
                    "KeyType": "HASH"
                },
                {
                    "AttributeName": "Actor",
                    "KeyType": "RANGE"
                }
            ],
            "Projection": {
                "ProjectionType": "ALL"
            },
            "ProvisionedThroughput": {
                "ReadCapacityUnits": 5,
                "WriteCapacityUnits": 5
            }
        }
    ],
    ProvisionedThroughput={
        'ReadCapacityUnits': 5,
        'WriteCapacityUnits': 5
    }
)
```

3.1.3. Using infrastructure-as-code with CloudFormation or Terraform

The last option is to use a strategy called infrastructure-as-code. With infrastructure-as-code, you declare the infrastructure you want to create in code. That code can be used in a repeatable way to create, modify, or delete your infrastructure.

While you could technically use the AWS CLI or AWS SDKs to manage infrastructure-as-code, it's more common to use a purpose-built tool for infrastructure-as-code. The most common tools are CloudFormation, which is provided by AWS, or Terraform, which is provided by Hashicorp, an independent developer tooling company.

An example of provisioning your table in CloudFormation is as follows:

```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
  MovieRolesTable:
    Type: AWS::DynamoDB::Table
    Properties:
      TableName: "MovieRoles"
      AttributeDefinitions:
        - AttributeName: "Actor"
          AttributeType: "S"
        - AttributeName: "Movie"
          AttributeType: "S"
      KeySchema:
        - AttributeName: "Actor"
          KeyType: "HASH"
        - AttributeName: "Movie"
          KeyType: "RANGE"
      ProvisionedThroughput:
        ReadCapacityUnits: "5"
        WriteCapacityUnits: "5"
      GlobalSecondaryIndexes:
        - IndexName: "ByMovie"
          KeySchema:
            - AttributeName: "Movie"
              KeyType: "HASH"
            - AttributeName: "Actor"
              KeyType: "RANGE"
          Projection:
            ProjectionType: "ALL"
          ProvisionedThroughput:
            ReadCapacityUnits: "5"
            WriteCapacityUnits: "5"
```

This has the same properties and values as our previous examples. The big difference here is that CloudFormation is written in YAML rather than a programming language like Bash or Python.

And this underlies a bigger difference between the last two approaches: infrastructure-as-code is usually *declarative*, in contrast with the imperative nature of using the CLI or an SDK.

With an imperative format, you're giving specific instructions on the actions to take. In our examples above, we used the CreateTable operation to create a table via the

CLI or SDK. If we wanted to change our table later, either to add a new secondary index or to change our billing properties, we would need to use the UpdateTable operation. The UpdateTable operation operates on an existing table and tells it to change specific attributes.

There is no CreateTable or UpdateTable operation in CloudFormation or Terraform. Rather, both tools have you *declare* the end result you want. In the example above, we declare a table with a secondary index. When we deploy that CloudFormation template, AWS will look to see if that table already exists. If it doesn't, it will create it. If it does exist, it will ensure the existing properties conform to what was submitted.

Later, if I add a secondary index to my table and re-submit the CloudFormation, AWS will perform the same operations. It will see that the table already exists but doesn't have one of the secondary indexes. It will perform the necessary operations to add the secondary index.

3.1.4. Why you should use infrastructure-as-code

I strongly recommend using infrastructure-as-code with a tool like CloudFormation or Terraform for provisioning your DynamoDB tables and related infrastructure for a few reasons.

First, infrastructure-as-code is **repeatable**. If you create your DynamoDB table in the AWS console, there's no way for you to ensure the exact same procedures if you need to

replicate that configuration in another environment or if your table is accidentally destroyed. You'd need to return to the AWS console and point and click again.

The same is true if you're running one-off AWS CLI or SDK commands. It's a pain-staking process to type out all of those parameters once to create your table. How fun will it be to do it again a second time?

That said, some people save their create-table commands in a `scripts/` directory in their code repository, giving them a halfway solution to infrastructure as code. While this is better than nothing, it's not as good as using a dedicated tool like CloudFormation or Terraform.

The second reason I prefer using a specific tool like CloudFormation or Terraform is because it's *legible*. By legible, I mean understandable and interpretable to someone who didn't write it.

As a CloudFormation user, it's pretty easy for me to come into a new company or repository, see their infrastructure-as-code in CloudFormation, and understand what's going on. There's one way to provision a CloudFormation table, and it will be consistent across services.

This isn't the same with hand-rolled operations using the AWS CLI or SDK. You may have to split it up into multiple commands, one to create the table and one to update a table. And there are a hundred different ways to do error handling, rate limiting, ordering, idempotence, etc. Each one is going to add complexity and increase the ramp-up

time for new developers.

3.2. Provisioning with CloudFormation

Now that you've seen my recommendation to use infrastructure-as-code, let's see some tips for using CloudFormation. The principles will be very similar for using Hashicorp's Terraform, so feel free to use that as well.

First we'll see some tips for provisioning your table. Then we'll look at how to manage your IAM resources.

3.2.1. Provisioning your table

We saw an example of what it looks like to provision a DynamoDB table in CloudFormation before, but let's walk through the whole experience. For this to work, you will need the AWS CLI installed on your machine and some AWS credentials configured in your environment.

First, let's create a basic table. Create a file called `template.yaml` with the following contents:

```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
  MovieRolesTable:
    Type: AWS::DynamoDB::Table
    Properties:
      TableName: "MovieRoles"
      AttributeDefinitions:
        - AttributeName: "Actor"
          AttributeType: "S"
        - AttributeName: "Movie"
          AttributeType: "S"
      KeySchema:
        - AttributeName: "Actor"
          KeyType: "HASH"
        - AttributeName: "Movie"
          KeyType: "RANGE"
      ProvisionedThroughput:
        ReadCapacityUnits: "5"
        WriteCapacityUnits: "5"
```

This is our basic MovieRoles table without any secondary indexes defined.

Let's deploy our stack by running the following command with the AWS CLI:

```
aws cloudformation create-stack \
  --template-body file://template.yaml \
  --stack-name MovieRoles
```

It will include a StackId of your created stack.

After a few minutes, check the status of your stack with the DescribeStacks call from the AWS CLI:

```
aws cloudformation describe-stacks \
  --stack-name MovieRoles
{
    "Stacks": [
        {
            "StackId": "arn:aws:cloudformation:us-east-
1:955617200811:stack/MovieRoles/98f418e0-6d1b-11ea-bd68-125f4521752f",
            "StackName": "MovieRoles",
            "CreationTime": "2020-03-23T15:33:08.628Z",
            "RollbackConfiguration": {},
            "StackStatus": "CREATE_COMPLETE",
            "DisableRollback": false,
            "NotificationARNs": [],
            "Tags": [],
            "EnableTerminationProtection": false,
            "DriftInformation": {
                "StackDriftStatus": "NOT_CHECKED"
            }
        }
    ]
}
```

The output should show that your stack was created successfully. You can check the status of your DynamoDB table with the DescribeTable command:

```
aws dynamodb describe-table \
  --table-name MovieRoles
{
    "Table": {
        "AttributeDefinitions": ...,
        "TableName": "MovieRoles",
        "KeySchema": ...,
        "TableStatus": "ACTIVE",
        "CreationDateTime": 1584977592.201,
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 5,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-east-
1:955617200811:table/MovieRoles",
        "TableId": "55f166db-9eac-446e-80d8-675916e12ba8"
    }
}
```

This will include a lot of information about your table. I've

truncated the output above. Notice that there are no secondary indexes on your table.

Now update your template to add our secondary index. Change the contents of your `template.yaml` to look like this:

```
AWSTemplateFormatVersion: "2010-09-09"
Resources:
  MovieRolesTable:
    Type: AWS::DynamoDB::Table
    Properties:
      TableName: "MovieRoles"
      AttributeDefinitions:
        - AttributeName: "Actor"
          AttributeType: "S"
        - AttributeName: "Movie"
          AttributeType: "S"
      KeySchema:
        - AttributeName: "Actor"
          KeyType: "HASH"
        - AttributeName: "Movie"
          KeyType: "RANGE"
      ProvisionedThroughput:
        ReadCapacityUnits: "5"
        WriteCapacityUnits: "5"
      GlobalSecondaryIndexes:
        - IndexName: "ByMovie"
          KeySchema:
            - AttributeName: "Movie"
              KeyType: "HASH"
            - AttributeName: "Actor"
              KeyType: "RANGE"
          Projection:
            ProjectionType: "ALL"
          ProvisionedThroughput:
            ReadCapacityUnits: "5"
            WriteCapacityUnits: "5"
```

Now let's update our CloudFormation stack with the following command:

```
aws cloudformation update-stack \
  --stack-name MovieRoles \
  --template-body file://template.yaml
```

Again, we should receive a response with a StackId.

After a few minutes, check the status of your stack again:

```
aws cloudformation describe-stacks \
  --stack-name MovieRoles
{
    "Stacks": [
        {
            "StackId": "arn:aws:cloudformation:us-east-
1:955617200811:stack/MovieRoles/98f418e0-6d1b-11ea-bd68-125f4521752f",
            "StackName": "MovieRoles",
            "CreationTime": "2020-03-23T15:33:08.628Z",
            "LastUpdatedTime": "2020-03-23T15:47:31.408Z",
            "RollbackConfiguration": {},
            "StackStatus": "UPDATE_COMPLETE",
            "DisableRollback": false,
            "NotificationARNs": [],
            "Tags": [],
            "EnableTerminationProtection": false,
            "DriftInformation": {
                "StackDriftStatus": "NOT_CHECKED"
            }
        }
    ]
}
```

The output should now indicate that the update is complete. Check the status of your DynamoDB table:

```
aws dynamodb describe-table \
  --table-name MovieRoles
{
    "Table": {
        "AttributeDefinitions": ...
        "TableName": "MovieRoles",
        "KeySchema": ...
        "TableStatus": "ACTIVE",
        "CreationDateTime": 1584977592.201,
        "ProvisionedThroughput": {
            "NumberOfDecreasesToday": 0,
            "ReadCapacityUnits": 5,
            "WriteCapacityUnits": 5
        },
        "TableSizeBytes": 0,
        "ItemCount": 0,
        "TableArn": "arn:aws:dynamodb:us-east-
1:955617200811:table/MovieRoles",
        "TableId": "55f166db-9eac-446e-80d8-675916e12ba8",
        "GlobalSecondaryIndexes": [
            {
                "IndexName": "ByMovie",
                "IndexStatus": "ACTIVE",
                "IndexArn": "arn:aws:dynamodb:us-east-
1:955617200811:table/MovieRoles/index/ByMovie"
                ...
            }
        ]
    }
}
```

Notice that there is now a `GlobalSecondaryIndexes` property in the response because the index has been added to the table. We didn't need to tell CloudFormation that a specific type of update was coming to the table. It simply read our template, compared it to the current state of our resources, and made the necessary updates.

Further, CloudFormation is idempotent. You could run the same UpdateStack operation on your template, and no further changes would be made. Because your template reflects the current state of the world, any further updates would be a no-op.

3.2.2. Handling your IAM resources

Now that we know how to provision DynamoDB tables in our CloudFormation, let's see how to handle our IAM resources as well. Remember in the last chapter that all authentication and authorization for DynamoDB is handled via AWS IAM. We can manage these resources via infrastructure-as-code as well, making it easy to update permissions, audit changes, and provide a repeatable deployment process.

In the last chapter, I said that IAM roles associated with your compute (EC2 instances or Lambda functions) are the ideal way to handle your application's DynamoDB permissions. Let's add those here.

Imagine you have a Lambda function that needs to run a PutItem operation against your MovieRoles table. You can add it to your CloudFormation by adding the following snippet to your CloudFormation template:

```
Resources:
  MovieRolesTable:
    Type: AWS::DynamoDB::Table
    Properties: ...
  LambdaFunctionRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: 2012-10-17
        Statement:
          - Effect: Allow
            Principal:
              Service:
                - lambda.amazonaws.com
            Action:
              - 'sts:AssumeRole'
      Policies:
        - PolicyName: AllowPutItemAccess
          PolicyDocument:
            Version: 2012-10-17
            Statement:
              - Effect: Allow
                Action:
                  - 'dynamodb:PutItem'
                Resource:
                  - Fn::GetAtt:
                    - MovieRolesTable
                    - Arn
```

I've truncated the `MovieRolesTable` resource to remove some of the properties, but make sure you leave that in there. Then append the `LambdaFunctionRole` resource below the `MovieRolesTable`.

There are a few important things to note about this new resource. First, notice that we're creating an `AWS::IAM::Role` resource. This is the IAM role we discussed that will be attached to our Lambda function.

Second, look at the `AssumeRolePolicyDocument` on the IAM role. It states that the principal to assume the role must be the service `lambda.amazonaws.com`. This ensures that *only* a Lambda function may assume the role. An IAM user or an EC2 instance will not be allowed to assume the

role.

Finally, look at the `Policies` property. I've attached one policy statement to the IAM role that allows for `dynamodb:PutItem` permissions on our MovieRoles table.

Look at the `Resource` declaration in the policy statement. I'm using a CloudFormation function called `Fn::GetAtt`. This allows me to get an attribute about a different resource created in my CloudFormation template. Remember that the `Resource` declaration in an IAM statement needs the ARN of a DynamoDB table, but my table is being created in this stack! This function allows me to retrieve the ARN once that DynamoDB table is created.

It is possible for us to construct the ARN since it is knowable, but it's better to parameterize this statement so the same template is reusable across accounts and regions. Further, we're less reliant on "magic strings" in our templates.

We can create the role with our CloudFormation UpdateStack command:

```
aws cloudformation update-stack \
  --stack-name MovieRoles \
  --template-body file://template.yaml \
  --capabilities CAPABILITY_IAM
```

Note that we've added a `capabilities` parameter to indicate we want to allow for creating IAM resources. CloudFormation makes you pass this explicitly to avoid unwanted IAM creation.

Imagine your role also had the need to run the Query operation on the ByMovie index in your table. We could update the CloudFormation as follows:

```
Resources:
  MovieRolesTable:
    Type: AWS::DynamoDB::Table
    Properties: ...
  LambdaFunctionRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: 2012-10-17
        Statement:
          - Effect: Allow
            Principal:
              Service:
                - lambda.amazonaws.com
            Action:
              - 'sts:AssumeRole'
      Policies:
        - PolicyName: AllowPutItemAccess
          PolicyDocument:
            Version: 2012-10-17
            Statement:
              - Effect: Allow
                Action:
                  - 'dynamodb:PutItem'
                Resource:
                  - Fn::GetAtt:
                    - MovieRolesTable
                    - Arn
        - PolicyName: AllowQueryOnByMoviesIndex
          PolicyDocument:
            Version: 2012-10-17
            Statement:
              - Effect: Allow
                Action:
                  - 'dynamodb:Query'
                Resource:
                  - Fn::Sub: "${Arn}/index/ByMovie"
                  - Arn:
                      Fn::GetAtt:
                        - MovieRolesTable
                        - Arn
```

We've added a second policy called `AllowQueryOnByMoviesIndex`. Most of the format is the same, but look again to the `Resource` block. There is no

way to use `Fn::GetAtt` to retrieve an *index's* ARN. Rather, we need to partially construct it ourselves. We use the `Fn::Sub` method to create a string composed of the table's ARN and a suffix of `/index/ByMovie`. Then we substitute in the table's ARN using the same `Fn::GetAtt` method we used above.

After you run the CloudFormation UpdateStack operation, your IAM role will then have Query permissions on your ByMovie index.

3.3. Gaps with infrastructure-as-code

CloudFormation is great for its repeatability, its legibility, and its idempotence. And infrastructure-as-code works very well with DynamoDB as compared to other database options. Not only can you create your database tables and indexes with CloudFormation, but you can also create all your IAM resources. It's a different story with a relational database where you can create the instance infrastructure, but you'll need to run a separate command in your database to create the tables, indexes, users, and permissions for your application to run.

That said, there are a few gaps in the CloudFormation experience around DynamoDB. Let's walk through those quickly.

- **Global tables:** DynamoDB Global Tables are a way to connect DynamoDB tables in multiple AWS regions and

replicate data across them. They work in a master-master pattern where writes in any region are replicated to all regions. This allows you to locate your data close to your users without managing replication yourself.

However, global tables don't work well with CloudFormation yet. All CloudFormation actions occur in a single region. If you want to configure global tables, you need to do this outside of CloudFormation.

- **Restoring from backups:** DynamoDB allows for backups and restores from backups. These are covered more thoroughly in Chapter 5 of this booklet. However, you cannot restore a DynamoDB table from a backup using CloudFormation. This means that if you do want to use a table that's restored from a backup, it will be managed outside of CloudFormation. There is some tooling to import resources into CloudFormation, but it's trickier than simply creating in CloudFormation in the first place.
- **Multiple index updates:** DynamoDB only allows for one secondary index to be added or updated at a time. Because of that, you can only add one secondary index per CloudFormation update. If you want to add two secondary indexes to a table, you must do this in two separate operations.

  Note that this limitation does not apply when creating a table, as you can create as many indexes as you want in a single operation when creating a new table.

3.4. Conclusion

DynamoDB table provisioning is an undercovered subject, likely because it's something you once do once at the beginning of your application. However, it is much easier to make updates later or to create additional environments if you do it correct at the outset.

I strongly recommend using an infrastructure-as-code tool like CloudFormation or Terraform to handle your DynamoDB resources. They're not that hard to learn, and they will save you a lot of time and sanity later on in your application's lifecycle.

Chapter 4. DynamoDB Pricing

Capacity planning has always been one of the more dreaded tasks of a new feature launch. The act of guessing how many users you'll get and how many requests they'll make is inaccurate enough. But then you descend into straight-up madness as you try to determine how to turn 'estimated number of concurrent operations' into CPU, RAM, and disk.

DynamoDB's pricing model is a huge improvement over traditional database pricing. Rather than paying for raw resources like CPU, you pay for actual read and write units. If you can accurately estimate your usage, you can accurately provision your capacity.

And what if you can't accurately estimate your usage? DynamoDB has you covered there too with a pay-per-use pricing model. Use this for spiky workloads or new applications where needs are still unclear.

In this chapter, we'll do a deep dive into DynamoDB pricing. We'll first cover how DynamoDB is priced and the different billing modes for operations. Then we'll discuss how to determine which billing mode you should use and provide a framework for thinking about DynamoDB cost. Finally, we'll cover some other ways to reduce your DynamoDB costs.

Pricing may feel like the domain of the finance team in the other office, but you'd be wise to consider it as a developer.

The cost of your DynamoDB table may determine whether you spend your time writing new features or toiling away with auto-scaling configurations and re-designs.

Let's start off with how DynamoDB pricing works before thinking about how to optimize your pricing strategy.

4.1. Basics of DynamoDB pricing

DynamoDB is priced on two axes: storage and operations. The storage axis is pretty straightforward. You will be charged a flat price per GB-month of all the items stored in your table. A GB-month refers to how much one GB of data would be charged for an entire month (744 hours). Some data will be in there the entire month and will be charged accordingly. Other data will be added during the month and will, therefore, consume less than an entire month.

Storage varies from $0.25/GB-month in regions like US East up to $0.375/GB-month in regions like South America. To reduce your storage price, you can write less data to DynamoDB or expire unneeded items from DynamoDB, whether manually or using DynamoDB's time-to-live (TTL) feature.

4.1.1. Operations pricing

The bigger part of DynamoDB pricing is in the operations. Before we get into the nitty-gritty on how operations

pricing works, let me say that this is one of my favorite parts of DynamoDB.

When you are doing capacity planning and costing for a relational database, you're flying blind. You can make estimates as to how many users you expect and the number of read and write operations as a part of that, but translating it to CPU, RAM, and disk requirements is another thing entirely. And so many factors affect the performance characteristics! How many rows are in your table? What about the table you're joining against? And what happens when the number of concurrent queries is 10x higher?

With DynamoDB, you're not doing guess work to translate queries into RAM. You are billed directly on read and write request *units*. And these units cost the same whether you're doing 100 requests per month or 100 million requests per minute, whether your database is 10 gigabytes or 10 terabytes.

A read request unit is when you make a *strongly-consistent* read of 4KB from your DynamoDB table. If you use *eventually-consistent* reads, you will be charged half, so a 4KB read only uses one-half read request unit. If you are using a *transactional* read, this is charged double, so a 4KB read consumes two read request units.

A similar calculation is used for write request units. When writing items up to 1KB in size, you will consume one write request unit. If you are doing a transactional write, you will consume double the write capacity units.

4.1.2. Two billing modes

For your operations, there are two different billing modes.

The traditional billing mode is provisioned capacity. With this, you set a specific number of read capacity units and write capacity units you want to provision. These numbers can be provisioned separately, so you can pay for 10,000 read capacity units and 100 write capacity units if you have a read-heavy workload. The provisioned capacity states how many capacity units you can consume *per second*. Thus 1000 read capacity units means you could read up to 8000 KB of data per second using eventually-consistent reads (1000 units * 4 KB per unit * 0.5 for eventually-consistent reads).

Provisioned capacity is paid for on an hourly basis, meaning you pay for the capacity whether you use it or not. Unused capacity does not carry over from period to period.

The newer billing mode is on-demand billing. Announced at AWS re:Invent 2018, on-demand billing lets you pay *per request* rather than paying for capacity whether you need it or not.

This is amazing. What other database options have a 'just pay for what you use' option? We've started to get used to this for things like stateless compute (Lambda) or blob storage (S3), but highly-available, single-digit millisecond data access that scales to terabytes? Insane.

Now, on-demand pricing comes at a premium. It costs about 7 times as much as *fully-utilized* provisioned capacity. However, as we'll see in the next section, it actually is the best pricing model for a wide variety of applications.

4.2. Which billing mode should I choose?

Now that you understand the two pricing models for DynamoDB operations, how do you determine which one to use?

As with many decisions, it really depends on your situation. Here are a few factors you should consider:

- **How well do you know your application traffic?** If you are launching a new service, you may have estimates for your application's usage but estimates are just that: estimates. It's hard to know what your real usage will be.

  On the other hand, if this application has been running in production for a year, you can look at historical metrics to see what your typical traffic patterns are.

- **How much will it cost to get a better estimate?** Even if you don't know your access patterns well, you could spend the time to model out different scenarios, add CloudWatch Alarms, and manually tune the capacity settings as needed. But this costs valuable time from expensive software engineers. Is this the best way to use their time?

- **How important is it to avoid throttling?** If you don't have enough capacity provisioned for a second in DynamoDB, your operations will be throttled, resulting in rejections. Think about how important it is to avoid throttling. If you can accept some level of throttling, you can be more aggressive about keeping costs down with provisioned capacity. If throttling is unacceptable, you'll want to overprovision or use on-demand pricing.
- **How variable is your usage?** Most applications have highly variable usage. Daytime hours use more traffic than nighttime hours, and weekday hours use more than weekend hours. It's possible your application will use many times more during your peak times as compared to your slow times. If you're using provisioned capacity, having enough capacity for your peak times will mean vastly over-paying during your slow times.

The variability of your usage is probably the biggest factor to consider. Imagine your traffic is 4 times higher during your peak periods as it is during your low periods. If you're doing pretty well with an average of 60% utilization in the high periods (which gives you room to handle temporary spikes!), this means you're only getting 15% utilization in the low periods. And I often see traffic differences of larger than 4X.

4.2.1. A framework for determining pricing model

With the factors above in mind, here's the framework I advise for most people.

First, if you are launching a brand new application with unknown traffic patterns, opt for on-demand pricing. It's unlikely you'll be able to estimate traffic accurately enough to outperform on-demand across both your high traffic times and low traffic times. Use on-demand pricing to establish a baseline, then re-evaluate in a few months once you have real data.

Second, is your application pretty low usage overall? If so, opt for on-demand, unless the provisioned capacity calculation is clear and straight-forward. It's rarely worth spending your time on it. If you have 200 read capacity units and 200 write capacity units per month, you're spending in the range of $100 per month. If you're having an engineer spend more than an hour or two on it, you're losing money.

Now, if you're in a position where you know the access patterns and it's a non-trivial amount of money, then it's a harder problem. You can take some time to run the numbers and choose the scenario best for you.

In general, I find that basic provisioned capacity is rarely the right choice, unless you have a pretty consistent workload. For most shops, the variability is enough to wash that out.

That said, a combination of provisioned capacity plus autoscaling can be the right choice. This can be Application Auto-Scaling as provided by AWS. If your traffic ramps up and down steadily, this can be a good fit.

If your usage is faster, you can do a pretty decent job with some rough heuristics. Set up a Lambda function that sets read capacity to 1500 units at 8AM and back down to 200 units at 8PM. It won't be perfect, but you'll get pretty decent utilization overall. You can tweak the parameters and sensitivity according to your situation and in line with the amount of money you're spending.

4.3. Other ways to keep your pricing down

When talking about a DynamoDB bill, people always want to focus on the billing models and different configuration values. This is the most readily apparent way to think you're making a difference with your bill.

But it's not always the best way to reduce your bill. Most of the time, you can make a big difference by *modeling your data correctly*. Remember that DynamoDB is billed based on the read and write capacity units you use. If you use fewer units, you can provision less capacity or pay for fewer on-demand requests, and your bill will go down.

With that in mind, let's review a few ways to reduce your usage.

4.3.1. Write efficient queries

The most effective way to reduce your spend is to write efficient queries. This is one of the main reasons I advise against using the Scan operation. In addition to being slow,

it often vastly overreads the data you need for your use case.

But even without the Scan operation, you can waste read capacity units. Are your Query operations narrowly targeted to retrieve only the items you need? Remember to use the primary key to do your filtering—any filtering you do in a filter expression will not reduce your read capacity consumption. You will still be charged for the full amount of the items read from the table, even if they're not returned to you.

4.3.2. Reduce write amplification

A second tip is to reduce write amplification across your secondary indexes. Write amplification is generally used when talking about relational databases to get a sense of how many indexes need to be updated when a row is written, which can use a lot of resources in your database. But there's a similar effect in DynamoDB. Remember that you must pay for write capacity for your secondary indexes separately from your base table. If you have five secondary indexes that must be updated each time you update an item in your base table, you'll need to pay 6 times (once on the table and five times for the indexes) for that write.

To reduce the cost here, think about how you can reduce write amplification. This could mean reducing or rethinking how often you update in your application. It could also mean updating the projection configuration in

your secondary indexes to only project those attributes that are needed by your access patterns. If you remove frequently-updated properties from your projection, you can greatly decrease the writes used.

Now this comes with a caveat. It can be really hard to manage projections in overloaded secondary indexes. Because you have a great diversity of attribute names across item types, you need to keep track of a large number of attributes. Further, DynamoDB limits the number of attribute names you can project. This strategy works best for indexes with few item types projected into it.

4.3.3. Split up items

A third tip is a read-based tip: split your large items into separate items. If you have a large item (>50KB), you're going to pay for all the read units every time you pull it back. It may be better to split the item into multiple smaller items based on how you'll access it. If most of your access patterns only need a few attributes on the item, put that into a specific item that can be retrieved directly. For access patterns that need the full item, you can retrieve those as needed.

Like the previous pattern, this adds complexity to your data model and application. I'd avoid it, unless it will make a big difference in your bill.

4.3.4. Temporary secondary indexes

The final tip I have is a favorite that I got from Rick Houlihan. Sometimes you may have an access pattern that is used infrequently. Perhaps it's a once-a-month query that reads from a sparse index to find specific types of items. However, if it's truly accessed that infrequently, you'll be overpaying by having a secondary index live the entire month handling all the updates.

Instead, you can create and destroy the secondary index on demand. When it's time for your monthly query, you can create your secondary index in the format needed by your query. When the index is ready, run your analytical queries and get your results. Then destroy the index. You'll pay for the writes needed to create the index, but you won't be paying for all the various updates throughout the month.

4.4. Conclusion

DynamoDB has a revolutionary billing model that is significantly better than other databases. Rather than buying resources and trying to guess how they translate into performance, DynamoDB guarantees you performance directly. You pay for your reads and writes without estimating how table size or concurrent queries affect your performance.

In this chapter, we learned about the different billing options for DynamoDB operations. We walked through

how to think about which billing mode you should choose and the factors to consider. Finally, we looked at some strategies for saving money on your AWS bill that don't involve billing configuration. Ultimately, proper data modeling goes a long way in keeping your DynamoDB bill down.

Chapter 5. DynamoDB Backup and Restore

Backups are a crucially important factor for a database. You need some way to ensure your data is not lost in the event of a catastrophic failure.

The way DynamoDB handles backups is amazing and underrated. Taking a backup doesn't consume write capacity on your table or affect table performance at all. Backups complete almost instantaneously and are stored securely in Amazon S3, separately from your table infrastructure. You can even enable constant backups with full point-in-time restoration to any second in the past 35 days. What other database offers anything like that?

In this chapter, we'll cover the ins and outs of DynamoDB backups and restores. We'll cover the two types of backup methods with DynamoDB and then discuss restoring your DynamoDB table from a backup.

5.1. Two types of backups

There are two backup mechanisms for DynamoDB tables. Let's discuss each of them to determine when you would choose one over the other.

5.1.1. On-demand backups

The first type of backup is the traditional on-demand backup. With an on-demand backup, you affirmatively reach out and tell DynamoDB to create a backup of your table at a particular time. This action can be done manually through the AWS console or via the AWS CLI, or it can be automated using some methods discussed below.

With an on-demand backup, DynamoDB will save a copy of your entire database in storage. And even though your table may contain terabytes of data, backups are nearly instantaneous. This is a pretty amazing feat. I recommend you watch the "Amazon DynamoDB Under the Hood" session from AWS re:Invent 2018 if you're interested in additional details on how that works.

The on-demand backup option costs $0.10 per GB-month of storage. Given the speed and mechanisms used to handle these updates, that's a fair price to pay.

With on-demand backups, you need to specifically trigger a backup. This is often automated through some system. I've used two different ways to automate this.

First, you can use AWS Backup to create a backup schedule for your table. AWS Backup is comparable to a cron service that focuses solely on backup operations. You set a schedule for your backups to be triggered, you add some targets to be included in your backup plan, and you let it run. I'd lean toward this option for most folks—it's simple and straightforward.

If you have more advanced needs, you can handle your own backups by writing an AWS Lambda function. This function can be triggered by a cron job or the occurrence of some sort of event. The Lambda function will have permission to create a backup on your DynamoDB table and will use the AWS SDK to trigger a backup.

5.1.2. Point-in-time recovery (PITR)

The second type of backup for DynamoDB is point-in-time recovery. When you enable point-in-time recovery on your DynamoDB table, DynamoDB will *continuously* maintain incremental backups of your table. You don't need to schedule an operation to trigger a backup.

Further, for tables with point-in-time recovery enabled, you can restore your DynamoDB to the exact point it was at for any second in the past 35 days. This is pretty amazing and becomes a very powerful disaster recovery tool. If you push some bad code that writes garbage data to your production table, you can restore your table to the exact second before that code was deployed. In contrast, if you were using on-demand backups, you would need to restore to the latest backup you created before pushing the bad code, which could result in significant data loss depending on the frequency of your backups.

Like on-demand backups, point-in-time recovery does not consume any write capacity units or affect performance on your DynamoDB table. Pricing for point-in-time recovery is $0.20 per GB-month, which is more expensive than on-

demand backups. However, as discussed below, it's likely to be cheaper for you overall.

5.1.3. Which backup mechanism to use?

Given these two backup mechanisms, which one should you use? I recommend point-in-time recovery for a few reasons.

First, it's completely 'set it and forget it'. You don't need to worry about scheduling backups and making sure your script is still working. Once you enable point-in-time recovery on your table, AWS will handle the rest.

Second, it's likely to be cheaper than on-demand backups, despite a storage cost that's twice as high. With an on-demand backup, you're likely to take backups on some sort of schedule—hourly, daily, weekly, etc. You will need to pay the $0.10/GB-month storage cost for *each* of those backups. In contrast, the point-in-time recovery cost is paid once for the entire table. This can make a big difference over time if you're creating a lot of backups.

Finally, the power of being able to restore to any second in the last 35 days is appealing. While I hope you never need to use this feature, it's comforting to know that you can reduce your data loss as much as possible in the event of a disaster.

That said, there may be a few times when on-demand backups are a better fit for you. The biggest one is around the backup period. Point-in-time recovery only stores the

state of your table for the last 35 days. If you need to store backups longer than that, you may need to look into on-demand backups.

Often I see people reach for on-demand backups for regulatory reasons. They may need to keep backups of their table for a period much longer than 35 days and thus use on-demand to handle that. In this situation, a combination of on-demand and point-in-time recovery may be best for you. You can use point-in-time for most disaster recovery scenarios but still make weekly or monthly on-demand backups of your table to handle your regulatory needs.

5.2. Restoring a table from a backup

Now that we've covered backups, let's handle the other part of disaster preparedness: recovering in the event of an issue.

Restores of a DynamoDB table from a backup are pretty easy. You'll likely use the AWS console to manually trigger a restore, as there's no way to create a DynamoDB table from a backup using CloudFormation.

When you restore a table, it will include the settings for your underlying table by default. However, you can change those settings before your restore if needed. This means you can not only change the billing mode or provisioned capacity units, but you can even remove secondary indexes

from your table before restoring. Removing secondary indexes can increase restore speed on your table.

Restore time is fairly quick. It won't be minutes if you have a large table, but we're talking in the range of hours, not days. Restore speed can be affected by the distribution of your partition key. Recall that a single DynamoDB partition can use a maximum of 1000 write capacity units. If you have data that is not well-distributed, your restore will be throttled as it tries to write to a partition with a lot of data.

One nice thing about restores is that you can do a cross-region restore. If you want to migrate your DynamoDB table to another region, this can be a faster and cheaper option than manually replicating those writes yourself. You'll need to ensure you track the writes on your table from the time you initiate the restore until it is restored and playback those writes on your restored table.

5.3. Conclusion

In this chapter, we discussed disaster management with DynamoDB. In keeping with DynamoDB's fully-managed nature, backups and restores are pretty low-maintenance operations. You can schedule on-demand backups to snapshot your table at a particular time, or you can enable point-in-time recovery to enable continuous backups of your DynamoDB table. If you need to restore, it's a relatively painless operation that gives you a fresh new

table as of the time of your restore.